HEAT 보고서

> 20191551 구형모

1장 HEAT 설계

옵션 1: 실패를 감지할경우 지정된 프로세스에 시그널 보내기

프로그램이 주어진 명령어를 실행하여 return 코드가 0이 아닐시, os.kill() 명령어를 통해 입력된 `--pid`와 \`--signal` 옵션을 가지고 `os.kill()` 함수를 통해 프로세스에 알맞은 신호를 보내도록 설계되었습니다. 이때 유효하지 않은 pid면 에러를 출력한후 종료합니다. 명령어는 python의 `subprocess.Popen()` 을 통해 실행되며 PIPE 로 stdout과 stderr가 연결됩니다.

옵션 2: 실패를 감지할경우 지정된 스크립트 실행하기

프로그램이 시작되면`os.access()` 함수를 활용하여 해당 script가 executable한지 확인하고, 그렇지 않으면 에러를 출력한후 종료합니다.

이후 명령어(check)의 return code가 0이 아닐경우, `subprocess.Popen()`함수인자에 `--fail` 옵션을 통해 제공된 script path를 넘기는 방식으로 실행합니다.

이때 다음과 같이 환경 변수가 설정됩니다.

- `HEAT_FAIL_CODE` 에 exit code를 전달
- `HEAT_FAIL_TIME` 에 발생시각(unixtime) 을 전달
- `HEAT_FAIL_INTERVAL` 에 인터벌 주기를 전달
- `HEAT_FAIL_PID` 에 실행한 ./check 의 PID를 전달

옵션 3: 실패가 누적될 경우 recovery 스크립트 지정하기

프로그램이 시작되면 `os.access()` 함수를 활용하여 해당 script가 executable한지 확인하고, 그렇지 않으면 에러를 출력한후 종료합니다. 또한 recovery 옵션을 통해 제공된 스크립트는 프로그램이 동기적으로 작동하기 때문에 도중에 fail 스크립트가 실행될 일은 없습니다. recovery 스크립트는 `subprocess.Popen()`함수인자에 `--recovery` 옵션을 통해 제공된 script path를 넘기는 방식으로 실행합니다.

실행이 끝난후 `subprocess.Popen()` 을 통해 검사 스크립트 혹은 명령어를 수행합니다. 이때, `time.time()`을 활용해 현재 unix time을 구한후, `--recovery-timeout`에 도달할때 까지 실행하는 방식으로 **지정된 시간**안에만 검사가 동작합니다. 만약 `--recovery-timeout` 이 없다면 `-`

`-threshold` 에서 **지정된 횟수** 만큼 검사 명령어나 스크립트를 반복합니다. 두 경우 모두 검사 명령어 혹은 스크립트가 실패 하면 fail 횟수만 누적 시킵니다.

만약 복구 되지 않는다면 다시 recovery를 호출합니다. 복구에 성공하면 fail 횟수를 초기화 하고 정상 모드로 진입 합니다.

recovery 스크립트 실행시 다음 환경 변수가 설정됩니다.

- `HEAT_FAIL_CODE` 에 exit code 를 전달
- `HEAT_FAIL_TIME` 에 최초 발생시각(unixtime) 을 전달
- `HEAT_FAIL_TIME_LAST` 에 최근 발생시각(unixtime) 을 전달
- `HEAT_FAIL_INTERVAL` 에 인터벌 주기를 전달
- `HEAT_FAIL_PID` 에 실행한 ./check 의 PID를 전달
- `HEAT_FAIL_CNT` 에 현재까지 연속 실패한 횟수를 전달

옵션 4: 실패가 누적될 경우 지정된 시그널 보내기

옵션3의 조건에 복구에 해당하는 경우 `--fault-signal` 옵션을 통해 지정된 시그널을 보거내거나, 옵션3의 복구 성공에 해당하는 경우 `--success-signal` 을 보냅니다. 이때, `os.kill()` 함수를 통해 `--pid` 옵션을 통해 지정된 프로세스에 알맞은 신호를 보내도록 설계되었습니다.

2장 HEAT 구현

1절 기본 구현

- python으로 구현되었습니다.
- 옵션들과 원격실행 명령은 argparse를 통해 런타임에서 파싱되며, 프로그램은 이를 분석하여 주어진 상황에 맞도록 실행됩니다 (heat.py 183줄 참조)
  - 아래 코드를 통해 명령어와 옵션은 args 전역 변수에 dictionary에 담기게 됩니다.
  - 이후 전역으로 선언된 dictionary에서 프로그램은 상황에 맞도록 읽어와 실행합니다.
  - ex) `heat -i 30 command` → `get_args()` → `option['i'] # 30`

```
def get_args():
    parser = argparse.ArgumentParser(add_help=False)
    parser.add_argument("-s")
    parser.add_argument("-i", type=int)
```

```
    parser.add_argument("--pid", type=int)
    parser.add_argument("--signal")
    parser.add_argument("--fail")
    parser.add_argument("--recovery")
    parser.add_argument("--threshold", type=int)
    parser.add_argument("--recovery-timeout")
    parser.add_argument("--fault-signal")
    parser.add_argument("--success-signal")

    parsed = parser.parse_known_args()
    return vars(parsed[0]), " ".join(parsed[1])
```

- `-s` 옵션을 통해 프로그램 실행시 스크립트의 executable 여부를 검사합니다(heat.py 168줄 참조)
  - 만약 명령어와 스크립트 둘다 주어진다면 `Error: use either script or command` 를 출력후 종료합니다.
  - `os.access()` 함수를 통해 executable 하지 않다면 `Failed: ./check not executable` 를 출력후 종료합니다

```
if args["s"]:
    if command:
        print("Error: use either script or command")
        exit(1)
    elif not os.access(args["s"], os.X_OK):
        print(f"Failed: {args['s']} not executable")
        exit(1)
```

- 명령어 실행 구현 (heat.py 139줄 참조)
  - `subprocess.Popen()`을 통해 명령어를 실행합니다. 이는 프로세스 생성을 의미 합니다
  - 생성된 stdout과 stderr는 PIPE 연결하였습니다
  - 반복문을 무한으로 돌리되, `-i`에서 주어진 인터벌 주기 만큼 `time.sleep()` 을 통해 프로세스 실행을 블록합니다
  - `wait()` 과 `terminate()` 로 동기화와 자원 회수를 진행합니다
  - 실행 시간은 `time.time()` 함수로 unix time을 불러옵니다. int형 타입캐스팅으로 정수로 변환합니다(기본적으로 실수)
  - 만약 return code 가 0이 아니라면 실패를 의미 하므로 `1669085130: Failed: Exit Code 22, details in heat.log` 과 같이 콘솔로 에러를 출력합니다. 성공한다면 `1669085130: OK` 와 같이 출력합니다

```
while True:
        res = subprocess.Popen(cmd,
                               stderr=subprocess.PIPE,
                               stdout=subprocess.PIPE,
                               shell=True,
                               universal_newlines=True)
        executed_time = int(time.time())
        res.wait()
        res.terminate()
        if res.returncode:
            print(f"{executed_time}:", f"Failed: Exit Code {res.returncode}, detai
ls in heat.log")
            on_post_error(executed_time, res.stderr)
            fail_cnt += 1
        else:
            print(f"{executed_time}: OK")

        if threshold and fail_cnt >= threshold:
            # fail 횟수를 기존에서 누적하고, 다시 recovery 를 호출한다.
            while True:
                if on_recovery(genesis_error_time=executed_time, check_pid=res.pi
d):  # 연속 누적 fail 횟수를 초기화하고 정상 모드로 진입
                    fail_cnt = 0
                    break
                fail_cnt += 1

        time.sleep(interval)
```

- 이때 heat.log 에는 좀 더 자세한 로그가 담기게됩니다. `open()` 함수를 통해 파일에 stderr PIPE로 부터 받아온 내용을 씁니다(heat.py 34줄 참조)

```
with open("heat.log", "a+") as file:
    while True:
        line = stderr.readline()
        if not line:
            break
        file.write(f"{executed_time}: {line}")
```

2절 옵션1: 실패를 감지할 경우 지정된 프로세스에 시그널 보내기

만약 실패를 감지(1절 옵션1 참조)한다면 `--pid` 옵션을 통해 제공된 프로세스 ID로 `--signal`을 통해 지정된 시그널을 보냅니다.

이때 해당 프로세스가 없으면 에러처리 후 종료하게 됩니다. (heat.py 41줄 참조)

```
if pid:
    if pid_exists(pid):
```

```
        sig_type = sig_name if sig_name else "HUP"
        send_signal(sig_type)
    else:
        print('올바른 PID를 입력해주세요')
        exit(1)
```

- 시그널이 없으면 SIGHUP을 전달합니다

프로세스가 존재하는지 확인하는 함수입니다.(heat.py 10줄 참조)

```
def pid_exists(pid):
    if pid <= 0:
        return False
    try:
        os.kill(pid, 0)
    except OSError as err:
        if err.errno == errno.EPERM:
            return True
        return False
    return True
```

- PID 값이 0 이하 이면 존재하지 않는 프로세스
- 해당 PID 값을 가진 프로세스에 시그널을 보내보고 시스템 에러가 호출되면 존재하지 않는 프로세스
- 단, 에러가 발생했음에도 EPERM 에러넘버는 권한 부족을 의미하며, 이는 곧 프로세스가 존재한다는 의미이므로 반례처리
- 해당 PID 값을 가진프로세스에 시그널 전송이 성공하면 존재하는 프로세스

프로세스 존재여부가 확인되면 다음 함수를 통해 시그널을 전송합니다.(heat.py 22줄 참조)

```
def send_signal(sig_type):
    pid = args["pid"]
    if pid and pid_exists(pid):
        os.kill(pid, signal.Signals[f"SIG{sig_type}"])
```

- `os.kill`을 통해 원하는 시그널 호출
- `signal.Signals`는 enum 객체로 `signal.Signals[”SIGKILL”]` 과 같이 호출하면 `9`를 반환합니다.

3절 옵션2: 실패를 감지할 실패를 감지할 경우 지정된 스크립트 실행하기

만약 `--fail` 옵션이 주어진다면 실패시 주어진 스크립트를 실행합니다. 이때 script의 excutable 여부를 판단하여 그렇지 않다면 `Failed: ./fail not executable` 를 콘솔에 출력하고 프로그램을 종료합니다.

```
if args["fail"] and not os.access(args["fail"], os.X_OK):
    print(f"Failed: {args['fail']} not executable")
    exit(1)
```

스크립트의 실행은 1절 기본 구현에서 프로세스를 설명하는 방식과 동일합니다. (heat.py 48줄 참조)

```
if fail_script_path:
    p = subprocess.Popen(fail_script_path,
                         stderr=subprocess.PIPE,
                         stdout=subprocess.PIPE,
                         shell=True,
                         universal_newlines=True)
    p.wait()
    p.terminate()
```

- `wait()` 함수를 통해 동기화를 진행하기 때문에 스크립트가 실행중 다음 인터벌을 수행하지 않습니다.

실패 함수를 실행할때는 `os.environ()` 함수를 통해 환경변수에 다음 값을 저장합니다 (heat.py 55줄 참조)

```
os.environ["HEAT_FAIL_CODE"] = f"{p.returncode}"
os.environ["HEAT_FAIL_TIME"] = f"{int(time.time())}"
os.environ["HEAT_FAIL_INTERVAL"] = f"{interval}"
os.environ["HEAT_FAIL_PID"] = f"{p.pid}"
```

4절 옵션3: 실패가 누적될 경우 recovery 스크립트 지정하기

만약에 검사 스크립트 혹은 명령어가 `--threshold` 옵션을 통해 지정된 정수값을 초과하여 연속 실패할경우 `--recovery` 옵션을 통해 지정한 스크립트가 실행됩니다.

이때 script의 excutable 여부를 판단하여 그렇지 않다면 `Failed: ./recover not executable` 를 콘솔에 출력하고 프로그램을 종료합니다.(heat.py 178줄 참조)

```
if args["recovery"] and not os.access(args["recovery"], os.X_OK):
    print(f"Failed: {args['recovery']} not executable")
    exit(1)
```

스크립트의 실행은 1절 기본 구현에서 프로세스를 설명하는 방식과 동일합니다. (heat.py 116줄 참조)

```
p = subprocess.Popen(recovery_script_path,
                     stderr=subprocess.PIPE,
                     stdout=subprocess.PIPE,
                     shell=True,
                     universal_newlines=True)
p.wait()
p.terminate()
```

- `wait()` 함수를 통해 동기화를 진행하기 때문에 스크립트가 실행중 fail 스크립트를 수행하지 않습니다.

recovery 스크립트의 실행이 끝나면 다음과 같이 동작합니다.

- `--recovery-timeout` 옵션에 지정된 시간안에 검사를 무한 루프로 돌린후, 지정된 간격(`-i`)마다 수행 (heat.py 75줄 참조)

```
started_time = int(time.time())
while True:
    if int(time.time()) - started_time > timeout:
        break
    p = subprocess.Popen(cmd,
                         stderr=subprocess.PIPE,
                         stdout=subprocess.PIPE,
                         shell=True,
                         universal_newlines=True)
    p.wait()
    p.terminate()
    if p.returncode == 0:
        return True # 복구 성공
    fail_cnt += 1  # 검사를 지정된 간격으로 계속 수행하고, fail 횟수만 누적
    time.sleep(interval)
return False # 복구 실패
```

  - `time.time()` 를 활용해 시작한 시간과 경과된 시간을 체크 하여 `--recovery-timeout` 을 벗어나는지 확인합니다.
  - 이후 return code가 0 이라면(성공) 반복문을 종료한후, 정상 모드로 진입합니다. (fail 횟수도 초기화), (heat.py 158줄 참조)

```
while True:
    if on_recovery():  # 연속 누적 fail 횟수를 초기화하고 정상 모드로 진입
        fail_cnt = 0
```

```
            break
        fail_cnt += 1
```

- 위 코드를 통해 복구되지 않고 지정된 시간이 지나면 다시 fail 횟수를 누적하고 recovery를 호출하게 됩니다.

- `--recovery-timeout` 이 없는 경우에는 `--threshold`에서 지정된 횟수동안 recover 스크립트가 동작합니다(heat.py 94줄 참조)

```
recovery_cnt = threshold
while recovery_cnt:
    recovery_cnt -= 1
    p = subprocess.Popen(cmd,
                         stderr=subprocess.PIPE,
                         stdout=subprocess.PIPE,
                         shell=True,
                         universal_newlines=True)
    p.wait()
    p.terminate()
    if p.returncode == 0:
        return True # 복구 성공
    fail_cnt += 1  # 검사를 지정된 간격으로 계속 수행하고, fail 횟수만 누적
return False # 복구 실패
```

  - 이후 동작은 위 `--recovery-timeout` 설명과 동일

recovery 스크립트 실행시 `os.environ()` 함수를 통해 환경변수에 다음 값을 저장합니다(heat.py 122줄 참조)

```
os.environ["HEAT_FAIL_CODE"] = f"{p.returncode}"
os.environ["HEAT_FAIL_TIME"] = f"{genesis_error_time}"
os.environ["HEAT_FAIL_TIME_LAST"] = f"{int(time.time())}"
os.environ["HEAT_FAIL_INTERVAL"] = f"{interval}"
os.environ["HEAT_FAIL_PID"] = f"{check_pid}"
os.environ["HEAT_FAIL_CNT"] = f"{fail_cnt}"
```

- genesis_error_time은 최초 발생시각을 의미하며 recover 실행 이전에 설정됩니다

5절 옵션4: 실패가 누적될 경우 지정된 시그널 보내기

프로세스 존재 여부를 체크하거나 시그널을 보내는 로직은 2절 옵션 1에 자세히 기술 되어 있으니 참조 부탁드립니다.

다음 조건일때에는 시그널을 보냅니다. 위에서 기술된 일부 코드를 생략 하였습니다.

- 옵션3의 조건에 복구에 해당하는 경우 fault-signal 을 보낸다(heat.py 71줄 참조)

```
if fault_signal:
    send_signal(fault_signal)
```

  - 실패후 recovery 호출 직전 실행
- 옵션3의 복구 성공에 해당하는 경우 success-signal 을 보낸다 (heat.py 88줄, 106줄 참조)

```
if timeout # recovery-timeout 옵션 사용
    while True:
        # 복구 실행
        if p.returncode == 0:
            if success_signal:
                send_signal(success_signal) # <-- here
            return True
    return False
else: # threshold 옵션 사용
    while recovery_cnt:
        # 복구 실행
        if p.returncode == 0:
            if success_signal:
                send_signal(success_signal) # <-- here
            return True
    return False
```

3장 HEAT 결과

강의 자료 기준인 Linux 기준으로 작성하였습니다. (단, AWS EC2 환경의 리눅스를 사용합니다)

1절 설치하기

파이썬 설치

파이썬이 이미 설치 되어 있다면 건너뛰어도 됩니다

```
$ which amazon-linux-extras
```

만약 아무것도 뜨지 않는다면 패키지를 설치해줍니다.

```
$ sudo yum install -y amazon-linux-extras
```

파이썬 설치를 활성해줍니다.

```
$ sudo amazon-linux-extras enable python3.8
```

Python 설치

```
$ sudo yum install python3.8 -y
```

이제 파이썬을 사용할 준비가 되었습니다.

pyinstaller 설치

pyinstaller를 통해 실행 가능한 bin 파일로 컴파일 합니다.

```
$ pip3 install pyinstaller
```

pyinstaller 가 설치되었습니다.

소스 컴파일 & bin 파일 생성

소스코드가 위치한 곳으로 이동합니다.

```
$ cd 경로
```

아래 명령어를 통해 heat bin 파일을 생성합니다.

```
$ pyinstaller heat.py --onefile
```

이제 dist/ 디렉토리내에 clsh 파일이 생성되었습니다. heat에 실행권한을 부여합니다. 다음을 통해 프로그램을 실행할 수 있습니다.

```
$ cd dist
$ chmod +x ./heat
$ ./heat [옵션] 명령어
```

2절 빠르게 시작하기

💡 heartbeat를 수신하는 간단한 서버(server.py)와 시그널을 지연시키고 수신된 시그널을 출력하는 간단한 프로그램(signal_receiver.py)을 구현해두었으며, 아래부터는 해당 서버와 프로그램을 활용하여 작성되었습니다.

server.py 실행

```
$ python3 server.py
```

```
❯ python3 server.py
Server listening on port 8000...
```

2초 주기로 CURL 명령어 실행하기

```
$ ./heat -i 30 curl -sf localhost:8000
```

```
❯ ./heat -i 2 curl -fs localhost:8000
1671306479: OK
1671306481: OK
1671306483: OK
1671306485: Failed: Exit Code 7, details in heat.log
1671306487: Failed: Exit Code 7, details in heat.log
```

실패 하게 된다면 heat.log 파일에서 상세한 에러를 볼 수 있습니다

```
1671308771:   % Total    % Received % Xferd  Average Speed   Time    Time     Time  Cu
rrent
1671308771:                                 Dload  Upload   Total   Spent    Left  Sp
eed
1671308771:
1671308771:0     0    0     0    0     0      0      0 --:--:-- --:--:-- --:--:--0
1671308771:0     0    0     0    0     0      0      0 --:--:-- --:--:-- --:--:--0
1671308771: curl: (7) Failed to connect to localhost port8000 after0 ms: Connection re
fused
```

쉘 스크립트를 지정하여 사용할 수도 있습니다

```
$ ./heat -i 30 -s ./check
```

```
❯ ./heat -i 2 -s ./check
1671308962: OK
1671308964: OK
1671308966: OK
```

3절 사용방법

옵션 1: 실패를 감지할 경우 지정된 프로세스에 시그널 보내기

프로그램은 실패를 감지하면 입력된 `--pid` 를 가진 프로세스에 `--signal` 로 입력된 시그널을 전송할 수 있습니다. 만약 시그널을 생략하면 SIGHUP이 전달됩니다.

```
$ ./heat -i 2 -s ./check --pid 46354 --signal USR1
```

```
❯ ./heat -i 2 -s ./check --pid 36737 --signal USR1
1671370006: OK
1671370008: OK
1671370010: Failed: Exit Code 7, details in heat.log
1671370013: Failed: Exit Code 7, details in heat.log
1671370015: Failed: Exit Code 7, details in heat.log
```

```
❯ python3 signal_receiver.py
running on 46354
Enter 'quit' to teminate>
Signal Recieved: SIGUSR1
Signal Recieved: SIGUSR1
Signal Recieved: SIGUSR1
```

옵션 2: 실패를 감지할 경우 지정된 스크립트 실행하기

프로그램은 `--fail` 옵션을 통해 실패를 감지하면 지정된 스크립트를 실행할 수 있습니다.

```
$ heat -i 30 -s ./check --fail ./failure.sh
```

```
❯ ./heat -i 2 -s ./check --fail ./fail
1671370576: Failed: Exit Code 7, details in heat.log
1671370581: Failed: Exit Code 7, details in heat.log
1671370586: Failed: Exit Code 7, details in heat.log
```

```
❯ ps -eo pid,command | grep fail
38331 ./heat -i 2 -s ./check --fail ./fail
38335 ./heat -i 2 -s ./check --fail ./fail
38524 /bin/bash ./fail
```

ps 명령어를 통해 fail script가 실행되고 있는지 확인

이후 다음 환경 변수에 값이 설정됩니다.

- `HEAT_FAIL_CODE` 에 exit code를 전달
- `HEAT_FAIL_TIME` 에 발생시각(unixtime) 을 전달
- `HEAT_FAIL_INTERVAL` 에 인터벌 주기를 전달
- `HEAT_FAIL_PID` 에 실행한 ./check 의 PID를 전달

옵션 3: 실패가 누적될 경우 recovery 스크립트 지정하기

프로그램은 -recovery 옵션을 통해 설정된 스크립트를 실행할 수 있습니다. 이때 --threshold를 통해 설정된 값이상 실패하게 되면 리커버리가 실행됩니다. 리커버리 스크립트 실행이 끝나면 다시 검사 스크립트 혹은 명령어를 실행 하는데 이를 --recovery-timeout 내에 순차적으로 시도하거나 --threshold 에서 지정된 값 만큼 동작합니다.

이후 검사결과가 성공이면 정상모드로 진입하지만, 복구되지 않고 지정된 시간 혹은 시도 횟수가 지나면 다시 리커버리를 진행합니다.

```
$ ./heat -i 2 -s ./check --recovery ./recover --threshold 3 --recovery-timeout 10
```

```
❯ ./heat -i 2 -s ./check --recovery ./recover --threshold 3 --recovery-timeout 10
1671372770: Failed: Exit Code 7, details in heat.log
1671372772: Failed: Exit Code 7, details in heat.log
1671372774: Failed: Exit Code 7, details in heat.log
```

```
  501 43724 43722   0 11:12PM ttys002    0:00.12 ./heat -i 2 -s ./check --recovery ./recover --threshold 3 --recovery-ti
meout 10
  501 43757 43724   0 11:12PM ttys002    0:00.02 /bin/bash ./recover
  501 43759 43757   0 11:12PM ttys002    0:00.08 python3 server.py
```

recovery 스크립트 실행시 다음 환경 변수 값이 설정됩니다.

- `HEAT_FAIL_CODE` 에 exit code 를 전달
- `HEAT_FAIL_TIME` 에 최초 발생시각(unixtime) 을 전달

- `HEAT_FAIL_TIME_LAST` 에 최근 발생시각(unixtime) 을 전달
- `HEAT_FAIL_INTERVAL` 에 인터벌 주기를 전달
- `HEAT_FAIL_PID` 에 실행한 ./check 의 PID를 전달
- `HEAT_FAIL_CNT` 에 현재까지 연속 실패한 횟수를 전달

옵션 4: 실패가 누적될 경우 지정된 시그널 보내기

프로그램은 `--pid` 에서 지정한 프로세스에 옵션 3에서 조건의 복구에 해당하는 경우 `--fault-signal` 을 보내거나, 옵션 3의 복구 성공에 해당하는 경우 `--success-signal` 을 보낼수 있습니다.

```
$ ./heat -i 2 -s ./check --pid 47001 --recovery ./recover --threshold 3 --recovery-tim
eout 10 --fault-signal USR1 --success-signal USR2
```

```
❯ ./heat -i 2 -s ./check --pid 47001 --recovery ./recover --threshold 3 --recovery-timeout 10 --fault-signal USR1 --succ
ess-signal USR2
1671374398: OK
1671374400: OK
1671374402: Failed: Exit Code 7, details in heat.log
1671374404: Failed: Exit Code 7, details in heat.log
1671374406: Failed: Exit Code 7, details in heat.log
1671374413: OK
1671374415: OK
1671374417: OK
```

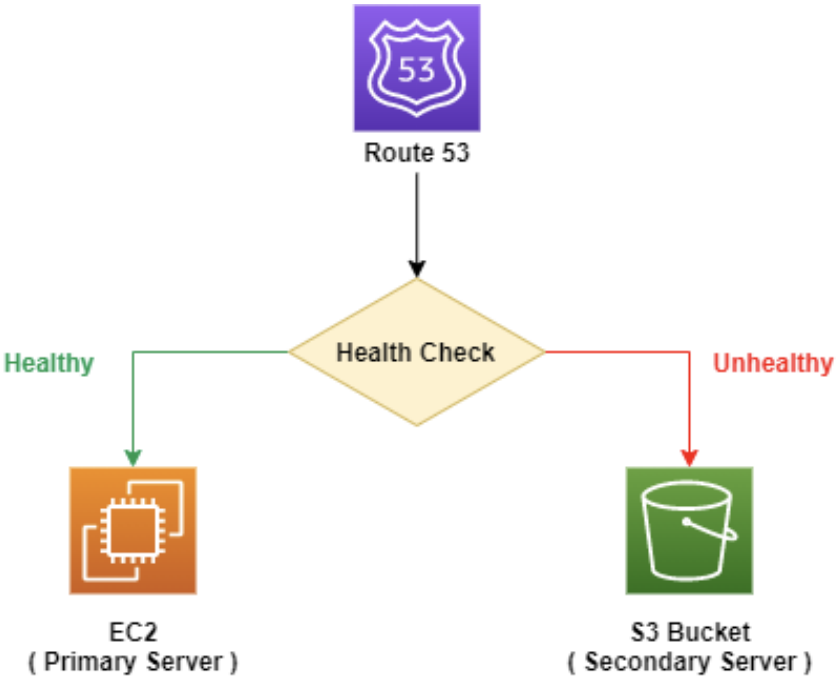
```
❯ python3 signal_receiver.py
running on 47001
Enter 'quit' to teminate>
Signal Recieved: SIGHUP
Signal Recieved: SIGHUP
Signal Recieved: SIGHUP
Signal Recieved: SIGUSR1
Signal Recieved: SIGUSR2
```

4장 적용 사례

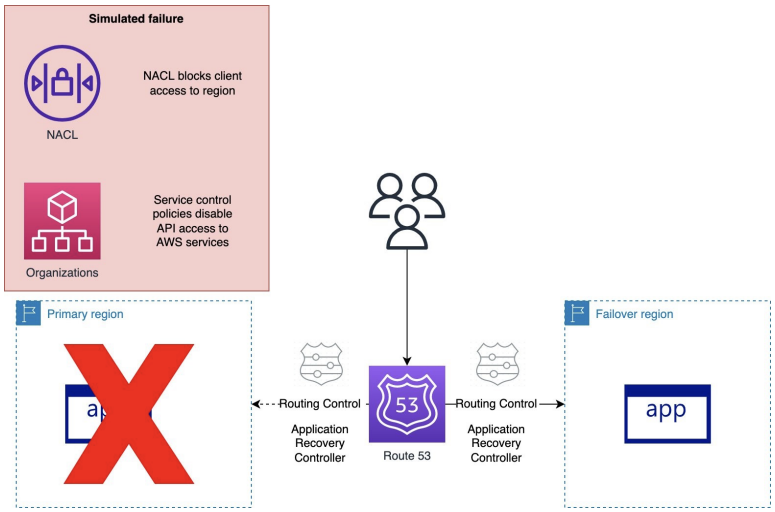
HEAT가 추상적으로 적용되어 있는 AWS의 Health Check의 경우를 예를 들 수 있습니다

Route53은 AWS DNS 웹서비스입니다. 내부적으로 Health Check 기능을 지원합니다.

아래 그림은 웹페이지 Serving를 예로 든 사진입니다. 고가용성을 유지하기 위해 인프라를 다음과 같이 설계할 수 있습니다. 특정 엔드포인트로 request를 보낸후(check 스크립트 실행), Healthy(서버로 연결 가능) 하다면 서버로 연결하지만, Unhealthy(서버가 죽음) 하다면 S3(정적호스팅 서비스)에 저장된 웹사이트를 불러와 서버를 살릴동안(recovery 스크립트 실행) 임시로 띄우도록 설계할 수 있습니다.



또 다른 예시 입니다. 서버가 미국에 위치하여 있는데 이곳에 장애가 생겨 유저가 접속하지 못하게 될 수도 있습니다. 이때 서버가 복구 될 동안(recovery script 실행) Primary Region 인 미국이 아닌 replica혹은 mirror 되어있는 Failover Region인 한국으로 접속을 우회 시킬 수 있습니다. 특정 시간 혹은 횟수동안 계속 primary 서버를 체크 해본후 다시 기존 region 으로 트래픽을 포워딩 할 수 있습니다.



이제 개발자는 HEAT과 유사한 이런 서비스 덕분에 24시간 컴퓨터 앞에 앉아있지 않아도 고가용성을 유지 할 수 있을 것 입니다.